

Machine Learning with Scikit-learn



David Mertz, Ph.D.
Anaconda Inc

Who am I?

August 2018

David was a Director of the PSF for six years, and remains co-chair of its Trademarks Committee, its Python-Cuba working group, and of the Scientific Python Working Group. Nowadays he is Senior Trainer and Senior Software Developer for Anaconda Inc.

He wrote the columns, *Charming Python* and *XML Matters* for IBM developerWorks, short books for O'Reilly, and the Addison-Wesley book *Text Processing in Python*, has spoken at multiple OSCon's, PyCon's, and at AnacondaCon, and was invited keynote speaker at PyCon-India, PyCon-UK, PyCon-ZA, PyCon Belarus, PyCon Cuba, and PyData SF.

Before Anaconda, David worked for 8 years with the folks (D. E. Shaw Research) who have built the world's fastest, highly-specialized (down to the ASICs and network layer), supercomputer for performing molecular dynamics. He is pleased to find Python has become the default high-level language for most scientific computing projects.

The latest version of this blurb will stay in a [Gist](https://gist.github.com/DavidMertz/e889b5d56008eb897cf9af73510c1901)
(<https://gist.github.com/DavidMertz/e889b5d56008eb897cf9af73510c1901>).



What is Anaconda?

Anaconda Inc is known as the creator of the Anaconda Distribution, Anaconda Enterprise, and as contributor or main sponsor of open source projects (some listed below):

Anaconda Distribution

With over 6 million users, the open source Anaconda Distribution is the fastest and easiest way to do Python and R data science and machine learning on Linux, Windows, and Mac OS X. It's the industry standard for developing, testing, and training on a single machine.

Anaconda Enterprise

Anaconda Enterprise is an AI/ML enablement platform that empowers organizations to develop, govern, and automate AI/ML and data science from laptop through training to production. It lets organizations scale from individual data scientists to collaborative teams of thousands, and to go from a single server to thousands of nodes for model training and deployment.

Open Source Projects

- *Conda* (<https://conda.io/docs/>).
- *Dask* (<https://dask.pydata.org/en/latest/>).
- *Dask-ML* (<https://dask-ml.readthedocs.io/en/latest/>).
- *Numba* (<https://numba.pydata.org/>).
- *Bokeh* (<https://bokeh.pydata.org/en/latest/>).
- *DataShader* (<http://datashader.org/>).
- *Intake* (<https://intake.readthedocs.io/en/latest/quickstart.html>).
- *HoloViews* (<http://holoviews.org/>).
- *PyViz* (<http://pyviz.org/>).



What is *scikit-learn*?

Scikit-learn provides a large range of algorithms in machine learning that are unified under a common and intuitive API. Most of the dozens of classes provided for various kinds of models share the large majority of the same calling interface. Very often—as we will see in examples below—you can easily substitute one algorithm for another with nearly no change in your underlying code. This allows you to explore the problem space quickly, and often arrive at an optimal, or at least *satisficing* approach to your problem domain or datasets.

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on *NumPy*, *SciPy*, and *matplotlib*
- Open source, commercially usable - BSD license

§

What is Machine Learning?

According to Wikipedia on 16:57, 4 August 2018 (https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=853415971).

Machine learning

From Wikipedia, the free encyclopedia

For the journal, see [Machine Learning \(journal\)](#).
"Statistical learning" redirects here. For statistical learning in linguistics, see [statistical learning in language acquisition](#).

Machine learning is a subset of [artificial intelligence](#) in the field of [computer science](#) that often uses statistical techniques to give [computers](#) the ability to "learn" (i.e., progressively improve performance on a specific task) with [data](#), without being explicitly programmed.^[1]

The name *machine learning* was coined in 1959 by [Arthur Samuel](#).^[2] Evolved from the study of [pattern recognition](#) and [computational learning theory](#) in [artificial intelligence](#),^[3] machine learning explores the study and construction of [algorithms](#) that can learn from and make predictions on [data](#)^[4] – such algorithms overcome following strictly static [program instructions](#) by making data-driven predictions or decisions,^[5] through building a [model](#) from sample inputs. Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or infeasible; example applications include [email filtering](#), detection of network intruders or malicious insiders working towards a [data breach](#),^[6] [optical character recognition](#) (OCR),^[7] [learning to rank](#), and [computer vision](#).

Machine learning is closely related to (and often overlaps with) [computational statistics](#), which also focuses on prediction-making through the use of computers. It has strong ties to [mathematical optimization](#), which delivers methods, theory and application domains to the field. Machine learning is sometimes conflated with [data mining](#),^[8] where the latter subfield focuses more on [exploratory data analysis](#) and is known as [unsupervised learning](#).^[5]^[9] Machine learning can also be unsupervised^[10] and be used to learn and establish baseline behavioral profiles for various entities^[11] and then used to find meaningful anomalies.

Within the field of [data analytics](#), machine learning is a method used to devise complex models and algorithms that lend themselves to prediction; in commercial use, this is known as [predictive analytics](#). These analytical models allow researchers, [data scientists](#), engineers, and analysts to "produce reliable, repeatable decisions and results" and uncover "hidden insights" through learning from historical relationships and trends in the data.^[12]

Machine learning and data mining

Problems [\[show\]](#)

- [Supervised learning](#) (classification · regression) [\[show\]](#)
- [Clustering](#) [\[show\]](#)
- [Dimensionality reduction](#) [\[show\]](#)
- [Structured prediction](#) [\[show\]](#)
- [Anomaly detection](#) [\[show\]](#)
- [Neural nets](#) [\[show\]](#)
- [Reinforcement learning](#) [\[show\]](#)

Theory [\[show\]](#)

Machine-learning venues [\[show\]](#)

[Glossary of artificial intelligence](#) [\[show\]](#)

Related articles [\[show\]](#)

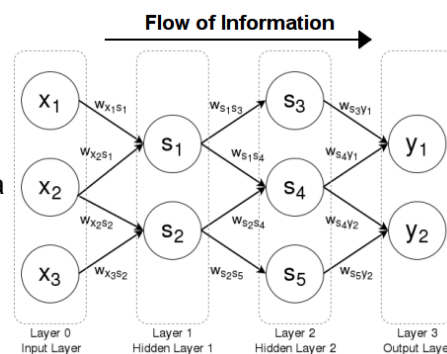
[Machine learning portal](#)

§

Difference between "Deep Learning" and other ML techniques

Neural Networks

The basic idea of a "multilayer perceptron" is a "feed-forward" artificial neural network, composed of "neurons" arranged in "layers." A common illustration is similar to that at right. This idea of "Hebbian networks" has existed since the 1940s, but it really only became a machine learning technique with Paul Werbos' 1975 introduction of "backpropagation" as a means to train such networks. Either way, the ideas are fairly old.



Included in diagram is a network with 4 layers and 12 connections (i.e. "parameters"). If it were "fully connected" the diagram would have 16 parameters. What makes a particular trained network special is the set of "weights" in the connections, illustrated and commonly named as subscripted w values.

For many decades after neural networks were known, they remained a minor area of interest. Usually a variety of other techniques rooted in statistics and linear algebra were more effective in solving problems of classification, regression, and clustering.

Image credit: "Feedforward Neural Networks", John McGonagle and yushi 21 (<https://brilliant.org/wiki/feedforward-neural-networks/>).

§

"Deep" really just means much larger...

In the last decade or less, neural networks—mathematically not much different from those described in the 1940s—grew much larger. For example, the extremely power Inception v3 image classifier consists of approximately 23.8 million parameters across about 140 layers. Layers generally each have many more neurons than the half-dozen or fewer shown in textbook illustrations like the one shown above. Scikit-learn has basic neural network techniques, but their use is mostly for the uses that made sense more than five years ago.

Classic "fully connected" layers make up only a small number of those used. More than anything else, the effect and reason for this is to limit the combinatorial explosion of connections, limiting the parameters to **only** 24 million.

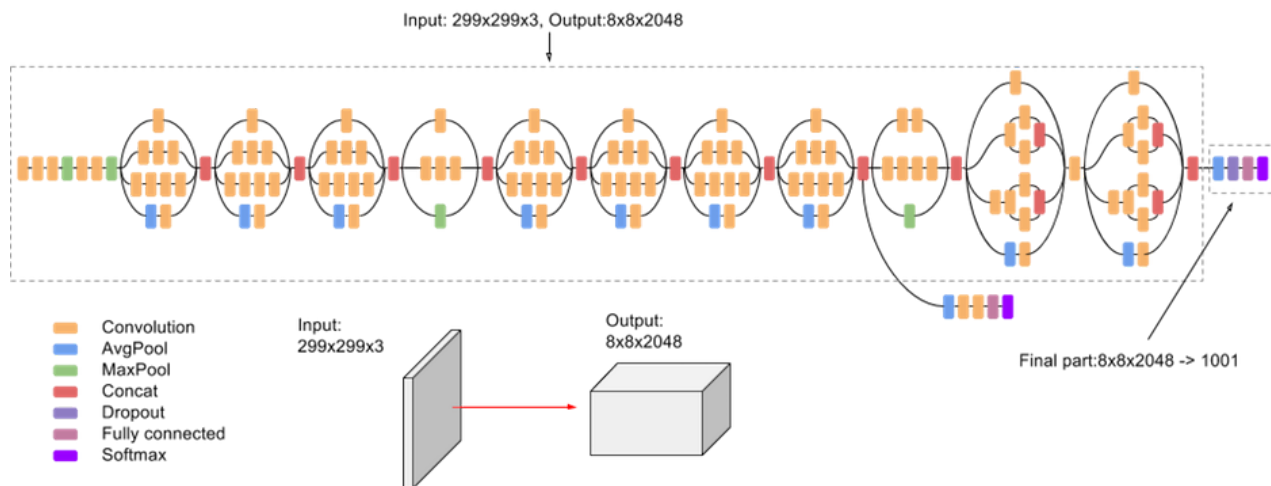


Image credit: ["Advanced Guide to Inception v3 on Cloud TPU" \(Google\)](https://cloud.google.com/tpu/docs/inception-v3-advanced)
(<https://cloud.google.com/tpu/docs/inception-v3-advanced>)

§

Other tools used for deep learning and computational requirements (GPU, etc)

Training modern deep neural networks requires large clusters of machines, ideally consisting of high-performance GPUs or even more specialized hardware like Google's Tensor Processing Unit (TPU). Competitive designs are impractical—or at least time consuming—to train on ordinary desktop systems or single servers. Evaluating a trained model is much cheaper, however, and running a classifier on ordinary higher-end CPUs is perfectly fine.

In contrast, the machine learning algorithms provided by Scikit-learn are generally easily trainable on ordinary computers, albeit not always entirely fast to train on typical machines.

Some popular tools for programming deep neural networks include the below. Python dominates in this space.

For Python:

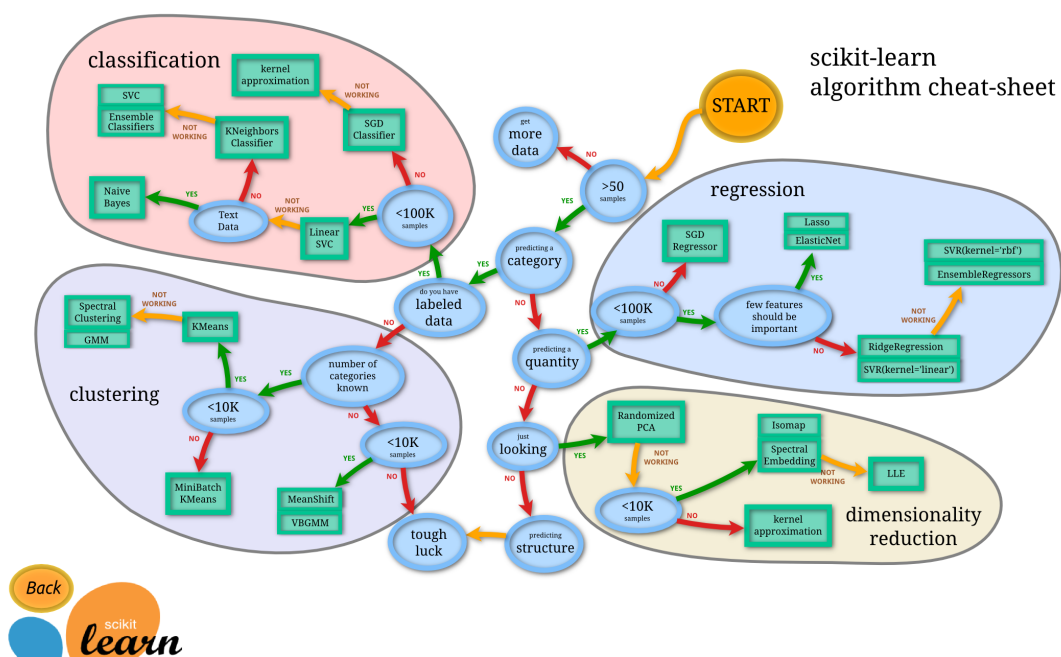
- Tensorflow
- Theano
- Keras
- Chainer
- PyTorch

For other programming languages:

- Torch
- Cafe2
- Microsoft Cognitive Toolkit (CNTK)
- Deep Learning 4J
- Apache MXnet

§

Techniques





"If you torture the data enough, nature will always confess." –Ronald Coase

Classification

Classification is a type of *supervised* learning in which the targets for a prediction are a set of categorical values.

Regression

Regression is a type of *supervised* learning in which the targets for a prediction are quantitative or continuous values.

Clustering

Clustering is a type of *unsupervised* learning where you want to identify similarities among collections of items without an *a priori* classification scheme. You may or may not have an *a priori* about the **number** of categories.

Dimensionality Reduction

Dimensionality reduction is most often a technique used to assist with other techniques. By reducing a large number of features to relatively few features; very often other techniques are more successful relative to these transformed synthetic features. Sometimes the dimensionality reduction itself is sufficient to identify the "main gist" or your data.



More Techniques

While the four techniques listed above are usually your ultimate goal in a machine learning design, some background concepts and/or techniques are often invaluable in getting there.

Feature Engineering

Very often, the "features" we are given in our original data are not those that will prove most useful in our final analysis. It is often necessary to identify "the data inside the data." Sometimes feature engineering can be as simple as normalizing the distribution of values. Other times it can involve creating synthetic features out of two or more raw features.

Feature Selection

Often, the features you have in your raw data contain some features with little to no predictive or analytic value. Identifying and excluding irrelevant features often improves the quality of a model.

Categorical vs. Ordinal vs. Continuous variables

Features come in one of three basic types.

Some are *categorical* (also called *nominal*): A discrete set of values that a feature may assume, often named by words or codes (but sometimes confusingly as integers where an order may be misleadingly implied).

Some are *ordinal*: There is a scale from low to high in the data values, but the spacing in the data may have little to no relationship to the underlying phenomenon. For example, while an airline or credit card "reward program" might have levels of Gold/Silver/Platinum/Diamond, there is probably no real sense in which Diamond is "4 times as much" as Gold, even though they are encoded as 1-4.

Some are *continuous* or *quantitative*: Some quantity is actually measured such that a number represents the amount of it. The *distribution* of these measurements is likely not to be uniform and linear (in which case scaling might be relevant), but there is a real thing being measured. Measurements might be quantized for continuous variables, but that does not necessarily make them ordinal instead. For example, we might measure annual rainfall in each town only to the nearest inch, and hence have integers for that feature.



Yet more techniques

Most of the concepts just discussed apply to statistics more generally. The next few concepts are more specific to *machine* learning.

One-hot encoding

For many machine learning algorithms, including neural networks, it is more useful to have a categorical feature with N possible values encoded as N features, each taking a binary value. Several tools, including a couple functions in *scikit-learn* will transform raw datasets into this format. Obviously, by encoding this way, dimensionality is increased.

Hyperparameters

The notion of *parameters* was introduced to define the way in which a model was trained. For neural networks, parameters are the weights of all the connections between the neurons. But in other models a similar parameterization exists. For example, in a basic linear regression, the coefficients in each dimension are parameters of the trained/fitted model.

However, many algorithms used in machine learning take "hyperparameters" that tune how the training itself occurs. These may be cutoff values where a "good enough" estimate is obtained, for example. Or there may be hidden terms in an underlying equation that can be set. Or an algorithm may actually be a family of closely related algorithms, and a hyperparameter chooses among them. Models in *scikit-learn* typically have a number of hyperparameters to set before they are trained (with "sensible" defaults when you do not specify).

Grid Search

While *scikit-learn* usually provides "sensible" defaults for hyperparameters, there is often a great deal of domain and dataset specificity for which hyperparameters are most effective. An API is provided to search across the combinatorial space of hyperparameter values and evaluate each collection.



Example: Machines Learning About Humans Learning About Machine Learning

I had the great honor and pleasure of presenting the first tutorial at [AnacondaCON 2018](https://anacondacon.io/) (<https://anacondacon.io/>), on machine learning with scikit-learn. I spoke to a full room of about 120 enthusiastic, and aspiring, data scientists.

The attendees of my session were a very nice group of learners and experts. But I decided I wanted to know even more about these people than I could find by looking at their faces and responding to their questions. So I asked them to complete a slightly whimsical form at about 3 hours into my tutorial. Just who are these people, and what can scikit-learn tell us about which of them benefitted most from the tutorial?

In the interest of open data science, the collection of answers given by attendees is available under a [CC-BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/legalcode) (<https://creativecommons.org/licenses/by-sa/4.0/legalcode>) license. Please credit it as "© Anaconda Inc. 2018" if you use the anonymized data, [available as a CSV file](https://goo.gl/WgTQMX) (<https://goo.gl/WgTQMX>). If you wish to code along with the rest of this notebook, save it locally as `Learning about Humans learning ML.csv` (or adjust your code as needed).

The attendees of this webinar, or those later viewing this notebook, are well described as:

"Humans learning about machines learning about humans learning about machine learning."



The whimsical dataset

Data never arrives at the workstation of a data scientist quite clean, no matter how much validation is attempted in the collection process. The respondent data is no exception. Using the familiar facilities in Pandas, we can improve the initial data before applying scikit-learn to it. In particular, I failed to validate the field "Years of post-secondary education (e.g. BA=4; Ph.D.=10)" as a required integer. Also, the "Timestamps" added by the form interface are gratuitous for these purposes—they are all within a couple minutes of each other, but the order or spacing is unlikely to have any value to our models.

In [1]:

```
import pandas as pd
import warnings
warnings.simplefilter("ignore")
```

In [2]:

```
fname = "Learning about Humans learning ML.csv"
humans = pd.read_csv(fname)

humans.drop('Timestamp', axis=1, inplace=True)
humans['Education'] = (humans[
    'Years of post-secondary education (e.g. BA=4; Ph.D.=10)']
    .str.replace(r'.*=', '')
    .astype(int))
humans.drop('Years of post-secondary education (e.g. BA=4; Ph.D.=10)',
    axis=1, inplace=True)
```

§

Eyballing data

When you read a dataset, it is almost always useful to take a quick look at part of it to get a "feel" for the data. If the dataset can be read as a Pandas DataFrame, overview inspection is particularly easy and friendly.

"90% of the time spent doing data analysis is doing data cleanup." –Every Data Scientist

In [3]:

humans.head(4)

Out[3]:

	Favorite programming language	Favorite Monty Python movie	Years of Python experience	Have used Scikit-learn	Age	In the Terminator franchise, did you root for the humans or the machines?	Which is the better game?	How successful has this tutorial been so far?
0	Python	Monty Python's Life of Brian	20.0	Yep!	53	Skynet is a WINNER!	Tic-tac-toe (Br. Eng. "noughts and crosses")	8
1	Python	Monty Python and the Holy Grail	4.0	Yep!	33	Team Humans!	Chess	9
2	Python	Monty Python and the Holy Grail	1.0	Yep!	31	Team Humans!	Chess	10
3	Python	Monty Python and the Holy Grail	12.0	Yep!	60	Team Humans!	Tic-tac-toe (Br. Eng. "noughts and crosses")	6

§

In terms of metadata *about* the dataset:

In [4]:

```
humans.describe(include=['object', 'int', 'float'])
```

Out[4]:

	Favorite programming language	Favorite Monty Python movie	Years of Python experience	Have used Scikit-learn	Age	In the Terminator franchise, did you root for the humans or the machines?	Which is the better game?
count	116	116	116.000000	116	116.000000	116	116
unique	7	6	NaN	2	NaN	2	4
top	Python	Monty Python and the Holy Grail	NaN	Yep!	NaN	Team Humans!	Chess
freq	94	57	NaN	80	NaN	88	69
mean	NaN	NaN	4.195690	NaN	36.586207	NaN	NaN
std	NaN	NaN	5.136187	NaN	13.260644	NaN	NaN
min	NaN	NaN	0.000000	NaN	3.000000	NaN	NaN
25%	NaN	NaN	1.000000	NaN	28.000000	NaN	NaN
50%	NaN	NaN	3.000000	NaN	34.000000	NaN	NaN
75%	NaN	NaN	5.000000	NaN	43.250000	NaN	NaN
max	NaN	NaN	27.000000	NaN	99.000000	NaN	NaN



§

Data cleanup

You can look into other features of the data yourself, but in the summary view a few data quality issues jump out. This is—again—almost universal to real world datasets. It seems dubious that two 3 year-olds were in attendance. Perhaps a couple 30-somethings mistyped entering their ages. A 99 year-old is possible, but seems more likely to be a placeholder value used by some respondent. While the description of what is meant by the integer "Education" was probably underspecified, it still feels like the -10 years of education is more likely to be a data entry problem than an intended indicator.

But the data we have is the data we must analyze.

Before we go further, it is usually a good idea to use [one-hot encoding](https://en.wikipedia.org/wiki/One-hot) (<https://en.wikipedia.org/wiki/One-hot>) of categorical data for machine learning purposes. Most likely this makes less difference for the decision tree and random forest classifiers used in this presentation than it might for other classifiers and regressors, but it rarely hurts. In this notebook, the encoding is performed with `pandas.get_dummies()` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html), but you could equally use `sklearn.preprocessing.LabelBinarizer` (<http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelBinarizer.html#sklearn.preprocessing.LabelBinarizer>) to accomplish the same goal.

In [5]:

```
human_dummies = pd.get_dummies(humans)
list(human_dummies.columns)
```

Out[5]:

```
['Years of Python experience',
 'Age',
 'How successful has this tutorial been so far?',
 'Education',
 'Favorite programming language_C++',
 'Favorite programming language_JavaScript',
 'Favorite programming language_MATLAB',
 'Favorite programming language_Python',
 'Favorite programming language_R',
 'Favorite programming language_Scala',
 'Favorite programming language_Whitespace',
 'Favorite Monty Python movie_And Now for Something Completely Different',
 'Favorite Monty Python movie_Monty Python Live at the Hollywood Bowl',
 'Favorite Monty Python movie_Monty Python and the Holy Grail',
 'Favorite Monty Python movie_Monty Python's Life of Brian',
 'Favorite Monty Python movie_Monty Python's The Meaning of Life',
 'Favorite Monty Python movie_Time Bandits',
 'Have used Scikit-learn_Nope.',
 'Have used Scikit-learn_Yep!',
 'In the Terminator franchise, did you root for the humans or the machine s?_Skynet is a WINNER!',
 'In the Terminator franchise, did you root for the humans or the machine s?_Team Humans!',
 'Which is the better game?_Chess',
 'Which is the better game?_Go',
 'Which is the better game?_Longing for the sweet release of death',
 'Which is the better game?_Tic-tac-toe (Br. Eng. "noughts and crosses")']
```

Classification: Choosing features and a target

It is time to use scikit-learn to model the respondents. In particular, we would like to know whether other features of attendees are a good predictor of how successful they found the tutorial. A very common pattern you will see in machine learning based on starting DataFrames is to drop one column for the X features, and keep that one for the y target.

In my analysis, I felt a binary measure of success was more relevant than a scalar measure initially collected as a 1-10 scale. Moreover, if the target is simplified this way, it becomes appropriate to use a **classification** algorithm as opposed to a **regression** algorithm. It would be a mistake to treat the 1-10 scale as a categorical consisting of 10 independent labels—there is something inherently ordinal about these labels, although *scikit-learn* will happily calculate models as if there is not. This is a place where subject matter judgement is needed by a data scientist.

You will have noticed by the summary data giving mean and median of success scores that ≥ 8 will approximately evenly divide the data into "Yes" and "No" categories.

In [6]:

```
X = human_dummies.drop("How successful has this tutorial been so far?", axis=1)
y = human_dummies["How successful has this tutorial been so far?"] >= 8
```

Conventional names and shapes

In almost all machine learning discussions, you will see the names capital-X and lowercase-y for the feature set and the target. The idea here is that the capital stands for the *independent variables*, but in general one expects there to be multiple such feature variables. The target consists of just one *dependent variable*, and hence its lowercase. The feature set and the target will always have the same number of rows.

Using X and y to distinguish independent and dependent variables is widespread in many areas of mathematics. Moreover, you will often see the features within X named x_1 , x_2 , x_3 , and so on in more academic texts.

§

In [7]:

```
y.head()
```

Out[7]:

```
0    True
1    True
2    True
3   False
4    True
```

Name: How successful has this tutorial been so far?, dtype: bool

In [8]:

```
X.iloc[:5,:5]
```

Out[8]:

	Years of Python experience	Age	Education	Favorite programming language_C++	Favorite programming language_JavaScript
0	20.0	53	12	0	0
1	4.0	33	5	0	0
2	1.0	31	10	0	0
3	12.0	60	10	0	0
4	7.0	48	6	0	0

§

Train/test split

While using `sklearn.model_selection.StratifiedKFold` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html) is a more rigorous way of evaluating a model, for quick-and-dirty experimentation, using `train_test_split` is usually the easiest approach. In either case, the basic principle is that you want to avoid *overfitting* by training on different data than you use to test your model.

In [9]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
print("Training features/target:", X_train.shape, y_train.shape)
print("Testing features/target:", X_test.shape, y_test.shape)
```

```
Training features/target: (87, 24) (87,)
```

```
Testing features/target: (29, 24) (29,)
```

§

Choosing an algorithm: Decision Trees and Random Forests

An interesting thing happened in trying a few models out. While `RandomForestClassifier` is incredibly powerful, and very often produces the most accurate predictions among all classifiers, for this particular data a single `DecisionTreeClassifier` does better. Attendees might want to think about why this turns out to be true and/or experiment with hyperparameters to find a more definite explanation; other classifiers might perform better still also, of course.

I will note that choosing the best `max_depth` for decision tree family algorithms is largely a matter of trial and error. You can search the space in a nice high level API using `sklearn.model_selection.GridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html), but it often suffices to use a basic Python loop like:

```
for n in range(1,20):
    tree = DecisionTreeClassifier(max_depth=n)
    tree.fit(X_train, y_train)
    print(n, tree.score(X_test, y_test))
```

In [10]:

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators=10, random_state=0)
rf.fit(X_train, y_train)
rf.score(X_test, y_test)
```

Out[10]:

0.5172413793103449

In [11]:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=7, random_state=0)
tree.fit(X_train, y_train)
tree.score(X_test, y_test)
```

Out[11]:

0.5862068965517241

§

Feature importances

Best practice in machine learning is to keep training and testing sets separate. In the end, with sufficiently large datasets, it makes little difference in the trained model parameters whether or how train/test observations are separated. But this is a small dataset, and also reflects a somewhat unique event (many students will learn about machine learning through many channels, but this particular tutorial, with a particular instructor, at a particular conference, will not necessarily generalize to all those channels).

Therefore, in order to see simply what is the "best possible" decision tree for this dataset, I deliberately overfit by including all the observations in the model.

Note that the feature importances identified by this classifier are not necessarily of the same order or ratios as would be produced by other classifiers.

In [12]:

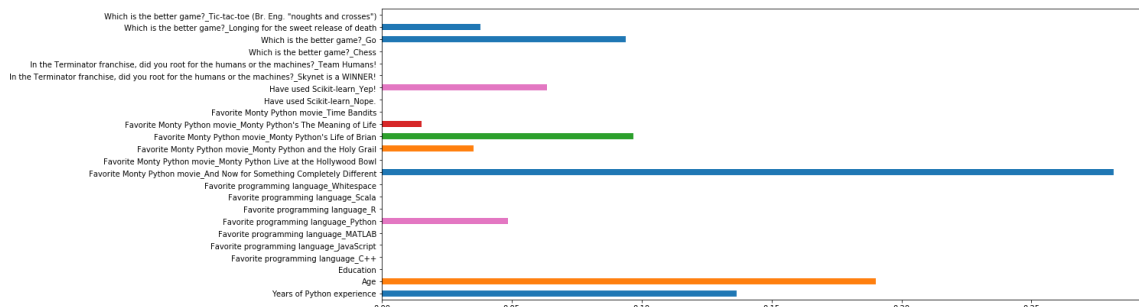
```
tree = DecisionTreeClassifier(max_depth=7, random_state=0)
tree.fit(X, y)
tree.score(X, y)
```

Out[12]:

0.8103448275862069

In [13]:

```
%matplotlib inline
pd.Series(tree.feature_importances_, index=X.columns).plot.barh(figsize=(18,7));
```



§

Cut points in a Decision Tree

As well as seeing what features are most important in this trained model, we can use a lovely utility method in `sklearn.tree` to display the entire tree and its decision cuts.

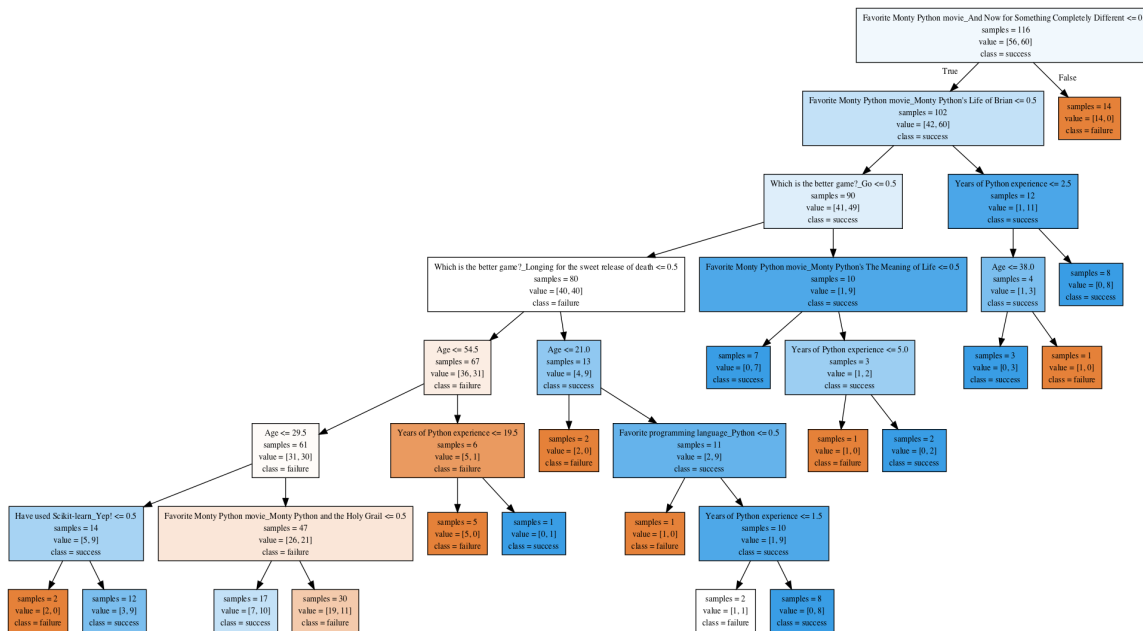
In [14]:

```
from sklearn.tree import export_graphviz
import sys, subprocess
from IPython.display import Image

export_graphviz(tree, feature_names=X.columns, class_names=['failure', 'success'],
                out_file='ml-good.dot', impurity=False, filled=True)
subprocess.check_call([sys.prefix+'/bin/dot', '-Tpng', 'ml-good.dot',
                      '-o', 'ml-good.png'])

Image('ml-good.png')
```

Out[14]:



In the diagram, blue branches reflect those respondents who found the tutorial more successful, and orange branches those who found it less so. The saturation of the displayed boxes reflects the strength of that decision branch.

As seems obvious in retrospect, the fans of *And Now for Something Completely Different* (https://en.wikipedia.org/wiki/And_Now_for_Something_Completely_Different) really did not like my tutorial very much. I probably should have provided a disclaimer at the beginning of the session. Years of Python experience is a slightly more important feature, but it follows an oddly stratified pattern wherein several different ranges of years show positive or negative effects—it's not linear.

And of course, *Time Bandits* (https://en.wikipedia.org/wiki/Time_Bandits) was not a Monty Python film at all: it is a Terry Gilliam film that happened to cast a number of Monty Python cast members. What on earth **were** those respondents thinking?!

§

Quick comparison of many classifiers in scikit-learn

"The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time." –Tom Cargill, Bell Labs

Having done the 90% of our work that was needed for data cleanup, the next 90% can be spent on model selection. Better hyper-parameters than those chosen below (mostly defaults) are likely to identify better fits. Moreover, the few classifiers listed are by no means all of those in *scikit-learn*.

This code is mostly based on the Scikit-learn [Classifier Comparison](http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html) (http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html).

Code source: Gaël Varoquaux & Andreas Müller
 Modified for documentation by Jaques Grobler
 License: BSD 3 clause

In [15]:

```
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

classifiers = {
    "KNN(3)"      : KNeighborsClassifier(3),
    "Linear SVM"   : SVC(kernel="linear"),
    "RBF SVM"     : SVC(gamma=2, C=1),
    "Gaussian Proc": GaussianProcessClassifier(1.0 * RBF(1.0)),
    "Decision Tree": DecisionTreeClassifier(max_depth=7),
    "Random Forest": RandomForestClassifier(max_depth=7, n_estimators=10, max_features=
4),
    "Neural Net"   : MLPClassifier(alpha=1),
    "AdaBoost"     : AdaBoostClassifier(),
    "Naive Bayes"  : GaussianNB(),
    "QDA"         : QuadraticDiscriminantAnalysis()
}
```

In [16]:

```
for name, classifier in classifiers.items():
    classifier.fit(X_train, y_train)
    score = classifier.score(X_test, y_test)
    print("{:<15}| score = {:.3f}".format(name, score))
```

KNN(3)	score = 0.517
Linear SVM	score = 0.621
RBF SVM	score = 0.586
Gaussian Proc	score = 0.483
Decision Tree	score = 0.517
Random Forest	score = 0.655
Neural Net	score = 0.552
AdaBoost	score = 0.552
Naive Bayes	score = 0.655
QDA	score = 0.276

§

Regression

For a regression, we want a range of target values, not a binary category. Having one-hot encoded binary features with a target that is more ordinal than continuous is close to the worst case for using a regression. It's time to move away from the "Learning about Humans learning ML" dataset. We will use a well-known example dataset that is included with *scikit-learn* for this next section.

§

Sample datasets

A number of sample datasets are included with *scikit-learn*, either already bundled with a `load_*`() function for the smaller ones or with a `fetch_*`() function for the larger ones that can be obtained online. The `make_*`() functions create synthetic datasets with some randomness in their generation.

In [17]:

```
from sklearn import datasets  
[attr for attr in dir(datasets) if not attr.startswith('_')]
```

Out[17]:

```
['base',  
 'california_housing',  
 'clear_data_home',  
 'covtype',  
 'dump_svmlight_file',  
 'fetch_20newsgroups',  
 'fetch_20newsgroups_vectorized',  
 'fetch_california_housing',  
 'fetch_covtype',  
 'fetch_kddcup99',  
 'fetch_lfw_pairs',  
 'fetch_lfw_people',  
 'fetch_mldata',  
 'fetch_olivetti_faces',  
 'fetch_rcv1',  
 'fetch_species_distributions',  
 'get_data_home',  
 'kddcup99',  
 'lfw',  
 'load_boston',  
 'load_breast_cancer',  
 'load_diabetes',  
 'load_digits',  
 'load_files',  
 'load_iris',  
 'load_linnerud',  
 'load_mlcomp',  
 'load_sample_image',  
 'load_sample_images',  
 'load_svmlight_file',  
 'load_svmlight_files',  
 'load_wine',  
 'make_biclusters',  
 'make_blobs',  
 'make_checkerboard',  
 'make_circles',  
 'make_classification',  
 'make_friedman1',  
 'make_friedman2',  
 'make_friedman3',  
 'make_gaussian_quantiles',  
 'make_hastie_10_2',  
 'make_low_rank_matrix',  
 'make_moons',  
 'make_multilabel_classification',  
 'make_regression',  
 'make_s_curve',  
 'make_sparse_coded_signal',  
 'make_sparse_spd_matrix',  
 'make_sparse_uncorrelated',  
 'make_spd_matrix',  
 'make_swiss_roll',  
 'mlcomp',  
 'mldata',  
 'mldata_filename',  
 'olivetti_faces',  
 'rcv1',  
 'samples_generator',  
 'species_distributions',
```

```
'svmlight_format',  
'twenty_newsgroups']
```



Boston Housing Prices

The Boston House Prices dataset contains a nice mixture of data. Most of it is continuous, a little bit is ordinal, and just one feature is a binary category. This contains a smallish number of rows, but is sufficient for illustration. The specific values you will see—especially in the house prices target—will seem quaintly 1970s; those who know Boston better will find that neighborhood demographics have changed in other aspects also.

In [18]:

```
boston = datasets.load_boston()  
print(boston.DESCR)
```

Boston House Prices dataset

=====

Notes

Data Set Characteristics:

:Number of Instances: 506

:Number of Attributes: 13 numeric/categorical predictive

:Median Value (attribute 14) is usually the target

:Attribute Information (in order):

- CRIM per capita crime rate by town
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS proportion of non-retail business acres per town
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX nitric oxides concentration (parts per 10 million)
- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940
- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per \$10,000
- PTRATIO pupil-teacher ratio by town
- B $1000(B_k - 0.63)^2$ where B_k is the proportion of blacks by town
- LSTAT % lower status of the population
- MEDV Median value of owner-occupied homes in \$1000's

:Missing Attribute Values: None

:Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.

<http://archive.ics.uci.edu/ml/datasets/Housing>

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics & Management, vol.5, 81-102, 1978. Used in Belsley, Kuh & Welsch, 'Regression diagnostics', Wiley, 1980. N.B. Various transformations are used in the table on pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression problems.

References

- Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
- Quinlan, R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.

- many more! (see <http://archive.ics.uci.edu/ml/datasets/Housing>)



Working with the example datasets

In [19]:

```
# Sample datasets can be easily converted to Pandas/Dask/PySpark DataFrames if desired
boston_df = pd.DataFrame(boston.data, columns=boston.feature_names)
boston_df.head()
```

Out[19]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396

In [20]:

```
# The X and y are stored in standard attributes
X = boston.data
y = boston.target
```



Comparing a gaggle of regressors

The metrics we use in the below code are `explained_variance_score`, `mean_absolute_error`, and `r2_score`. Many other metrics are available, mostly within the `sklearn.metrics` submodule.

In [21]:

```
from sklearn.metrics import explained_variance_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
```

The particular regressors we choose does not reflect any deep decision. Most are somewhat in the family of linear regression. RANSAC is tried because it is meant to be more resilient against outliers in data. The are *sometimes* more strongly predictive and generic linear regression. One of several Guassian techniques is shown as an example—it behaves worthlessly for this example, at least without hyperparameter tuning.

In [22]:

```
from sklearn.model_selection import cross_val_predict

from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression, RANSACRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.svm import SVR
from sklearn.svm import LinearSVR

regressors = [
    LinearRegression(),
    RANSACRegressor(),
    GaussianProcessRegressor(),
    KNeighborsRegressor(n_neighbors=9, metric='manhattan'),
    SVR(),
    LinearSVR(),
    SVR(kernel='linear') # Cf. LinearSVR: much slower, might be better or worse:
]

for model in regressors:
    predictions = cross_val_predict(model, X, y, cv=10)
    print(model)
    print("\tExplained variance:", explained_variance_score(y, predictions))
    print("\tMean absolute error:", mean_absolute_error(y, predictions))
    print("\tR2 score:", r2_score(y, predictions))
```

```

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
    Explained variance: 0.5901906801642502
    Mean absolute error: 4.002671788359303
    R2 score: 0.5901773933319251
RANSACRegressor(base_estimator=None, is_data_valid=None, is_model_valid=None,
    loss='absolute_loss', max_skips=inf, max_trials=100,
    min_samples=None, random_state=None, residual_metric=None,
    residual_threshold=None, stop_n_inliers=inf, stop_probability=0.9
9,
    stop_score=inf)
    Explained variance: 0.14621206921009655
    Mean absolute error: 5.450198720833523
    R2 score: -0.010140369273307526
GaussianProcessRegressor(alpha=1e-10, copy_X_train=True, kernel=None,
    n_restarts_optimizer=0, normalize_y=False,
    optimizer='fmin_l_bfgs_b', random_state=None)
    Explained variance: 0.0020213189051094815
    Mean absolute error: 22.523643045019917
    R2 score: -6.0074215059423235
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='manhattan',
    metric_params=None, n_jobs=1, n_neighbors=9, p=2,
    weights='uniform')
    Explained variance: -0.03306039129462701
    Mean absolute error: 6.451119894598156
    R2 score: -0.03319496816578171
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
    kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
    Explained variance: -0.060103421633902254
    Mean absolute error: 6.830978864168416
    R2 score: -0.08016044387841803
LinearSVR(C=1.0, dual=True, epsilon=0.0, fit_intercept=True,
    intercept_scaling=1.0, loss='epsilon_insensitive', max_iter=1000,
    random_state=None, tol=0.0001, verbose=0)
    Explained variance: 0.177281242274696
    Mean absolute error: 6.559715433114566
    R2 score: 0.15308192119408548
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
    kernel='linear', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
    Explained variance: 0.6180707121101677
    Mean absolute error: 3.7465111290105337
    R2 score: 0.6081908969664971

```



Hyperparameters

As mentioned earlier, models in *scikit-learn* have hyperparameters that configure their behavior. These can be set with named arguments to the class initializer and can be examined or set using attributes of the model object. For example, the `KNeighborsRegressor` used in the list above performed terribly, but perhaps that is because it has less good hyperparameters.

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='manhattan',
                    metric_params=None, n_jobs=1, n_neighbors=9, p=2,
                    weights='uniform')
Explained variance: -0.03306039129462701
Mean absolute error: 6.451119894598156
R2 score: -0.03319496816578171
```

In practice, you are more likely to start with an algorithm that does moderately well naively, then fine tune it. In this example, we start with one that is terrible (on the specific dataset; it does well elsewhere), and are simply trying to get it to less terrible for the example.



Grid Search

At heart a grid search is simply a number of nested loops, with the same combinatorial complexity as those loops make obvious. If we want to search over `KNeighborsRegressor` with procedural code, we could write:

```
for n_neighbors in [5,6,7,8,9,10,11,12]:
    for weights in ['uniform', 'distance']
        for metric in ['minkowski', 'chebyshev', 'manhattan']:
            knr = KNeighborsRegressor(n_neighbors=n_neighbors, weights=weights,
metric=metric)
            knr.fit(...)
            score = knr.score(...)
            # ... keep track of best score and hyperparameters ...
            # ... etc ...
```

The preferred API in *scikit-learn* is more intuitive and provides a few convenience methods on the searched objects (mostly the same methods as models themselves, in fact).



It looks like the best one can do with the Boston Housing dataset using `KNeighborsRegressor` remains terrible, but at least we can eliminate that technique for that data.

In [23]:

```
from sklearn.model_selection import GridSearchCV
knr = KNeighborsRegressor()

parameters = {'n_neighbors': [5,6,7,8,9,10,11,12],
              'weights': ['uniform', 'distance'],
              'metric': ['minkowski', 'chebyshev', 'manhattan']}

grid = GridSearchCV(knr, parameters)
model = grid.fit(X, y) # Best fit over cross-product of parameter space
print(model)
```

```
GridSearchCV(cv=None, error_score='raise',
            estimator=KNeighborsRegressor(algorithm='auto', leaf_size=30, metri
c='minkowski',
            metric_params=None, n_jobs=1, n_neighbors=5, p=2,
            weights='uniform'),
            fit_params=None, iid=True, n_jobs=1,
            param_grid={'n_neighbors': [5, 6, 7, 8, 9, 10, 11, 12], 'weights':
['uniform', 'distance'], 'metric': ['minkowski', 'chebyshev', 'manhatta
n']},
            pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
            scoring=None, verbose=0)
```

In [24]:

```
predictions = cross_val_predict(model, X, y, cv=10)
print("Explained variance:", explained_variance_score(y, predictions))
print("Mean absolute error:", mean_absolute_error(y, predictions))
print("R2 score:", r2_score(y, predictions))
```

```
Explained variance: -0.10094529567875443
Mean absolute error: 6.6938785787342585
R2 score: -0.10168975208105446
```

§

Clustering

There are about a dozen clustering algorithms in *scikit-learn*. Basically, the purpose of all of them is to find the "lumps" in a dimensional dataset where you do not have *a priori* labels. For no special reason, we chose SpectralClustering below.

In [25]:

```
from sklearn import cluster
spectral = cluster.SpectralClustering(
    n_clusters=4, eigen_solver='arpack',
    affinity="nearest_neighbors")
```

The API for clustering is as close to those for other models as is feasible. Specifically, these models also have a `.fit()` method, but one that does not need or accept targets.

In [26]:

```
spectral.fit(boston.data)

# Let us do minor Pandas manipulation to view clusters in an intuitive way
boston_df['category'] = spectral.labels_
boston_df['price'] = boston.target
house_clusters = boston_df.groupby('category').mean().sort_values('price')
house_clusters.index = ['low', 'mid_low', 'mid_high', 'high']
house_clusters[['price', 'CRIM', 'RM', 'AGE', 'DIS']]
```

Out[26]:

	price	CRIM	RM	AGE	DIS
low	12.800000	15.689918	6.103088	89.835294	2.000068
mid_low	17.418447	11.082796	5.974233	90.011650	2.072428
mid_high	21.212500	0.894798	6.087675	82.507500	2.480024
high	25.866090	0.248699	6.471142	54.576817	4.984178

§

Where to go next?



Explore Anaconda Enterprise:

<https://www.anaconda.com/enterprise/>

Schedule an Anaconda Enterprise Demo:

ambassador@anaconda.com

Learn about consulting, training, and support:

ambassador@anaconda.com



Table of Contents

- [Machine Learning with Scikit-learn](#)
- [Who am I?](#)
- [What is Anaconda?](#)
- [What is *scikit-learn*?](#)
- [What is Machine Learning?](#)
 - [Difference between "Deep Learning" and other ML techniques](#)
 - [Neural Networks](#)
 - ["Deep" really just means much larger...](#)
 - [Other tools used for deep learning and computational requirements \(GPU, etc\)](#)
- [Techniques](#)
 - [Classification](#)
 - [Regression](#)
 - [Clustering](#)
 - [Dimensionality Reduction](#)
- [More Techniques](#)
 - [Feature Engineering](#)
 - [Feature Selection](#)
 - [Categorical vs. Ordinal vs. Continuous variables](#)
- [Yet more techniques](#)
 - [One-hot encoding](#)
 - [Hyperparameters](#)
 - [Grid Search](#)
- [Example: Machines Learning About Humans Learning About Machine Learning](#)
 - [The whimsical dataset](#)
 - [Eyballing data](#)
 - [Data cleanup](#)
- [Classification: Choosing features and a target](#)
 - [Conventional names and shapes](#)
- [Train/test split](#)
 - [Choosing an algorithm: Decision Trees and Random Forests](#)
- [Feature importances](#)
- [Cut points in a Decision Tree](#)
- [Quick comparison of many classifiers in scikit-learn](#)
- [Regression](#)
 - [Sample datasets](#)
 - [Boston Housing Prices](#)
 - [Working with the example datasets](#)
 - [Comparing a gaggle of regressors](#)
- [Hyperparameters](#)
- [Grid Search](#)
- [Clustering](#)
- [Where to go next?](#)